GitHub Actions & Pure Docker 카나리 배포 완벽 가이드

비전공자도 완벽히 이해하는 로컬 무중단 배포 시스템 구축

Presentation Agenda

- **Concept:** 카나리 배포란 무엇인가?
- **Step 1:** Ubuntu 러너 컨테이너 띄우기
- **Step 2:** 컨테이너 환경 세팅 및 권한 부여
- **Step 3:** GitHub Runner 다운로드 및 설정
- **Step 4:** Nginx 및 App 웹 서비스 코드 작성
- **Step 5:** 순수 도커 카나리 스크립트 작성
- **Step 6:** GitHub Actions 지시서 작성
- **Step 7:** 초기 네트워크 세팅 및 Git 연동
- **Troubleshooting:** 잦은 오류 해결법
- **Demo:** 무중단 트래픽 전환 직접 관찰

The Concept

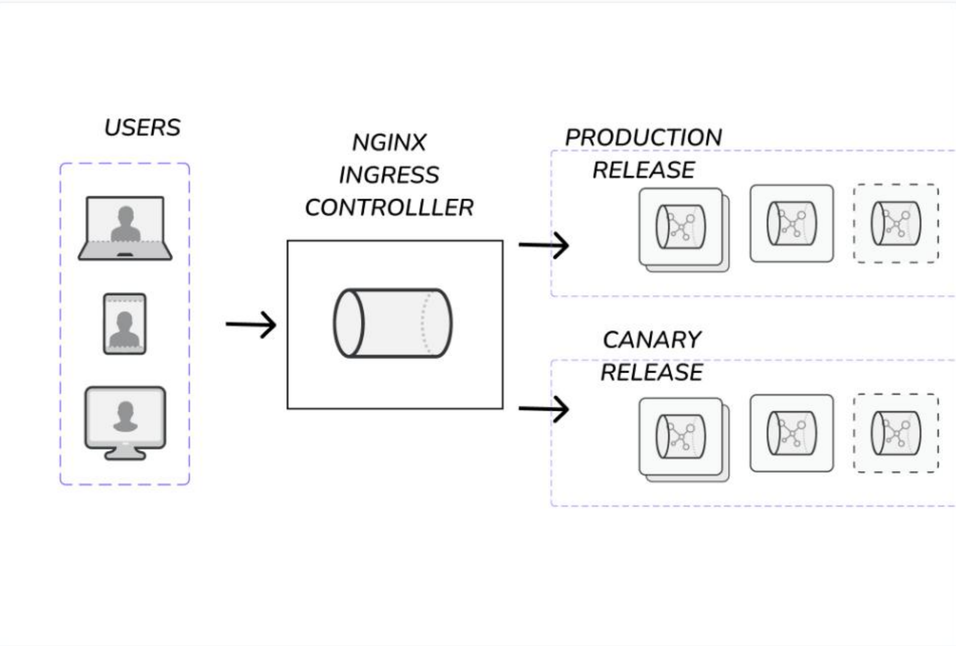
카나리(Canary) 배포 전략의 이해

카나리(Canary) 배포란?

광산의 유독 가스 감지용 **'카나리아' 새**에서 유래된 배포 명칭입니다.

핵심 원리

새 버전을 배포한 뒤, 전체 트래픽의 **10%**만 먼저 신규 버전으로 인입시킵니다. 모니터링 후 이상이 없으면 **50% → 100%**로 점진 상향하는 안전한 방식입니다.



카나리 배포 vs Blue/Green 배포

구분	Blue/Green 배포	Canary 배포
트래픽 전환	0%에서 100%로 **한 번에 일괄 전환**	10% → 50% → 100% **점진적 비율 전환**
위험도 (에러 발생)	모든 사용자가 동시에 서비스 장애 경험	소수의 사용자(10%)만 영향을 받음 (안전)
롤백 속도	즉각적인 라우팅 변경으로 매우 빠름	장애 감지 시 즉각 롤백(리스크 최소화 강점)

왜 순수 Docker인가요? (No Compose)

기초 원리 체득

자동화 도구에 의존하지 않고 컨테이너 명령어의 본질적 동작 원리를 학습합니다.

네트워크 이해

Docker Network를 직접 생성하여, 컨테이너 간 내부 DNS 통신 원리를 명확히 파악합니다.

생명주기 제어

Build, Run, Rm 명령어로 서버 생성부터 완전한 파기까지 생명주기를 통제합니다.

Step 1

순정 Ubuntu 러너 컨테이너 구동

Runner 백그라운드 구동 명령어

터미널을 열고, 단 한 줄의 강력한 명령어로 텅 빈 Ubuntu 서버를 백그라운드에 가동합니다.

```
docker run -d --name local-ubuntu-runner \ -v
/var/run/docker.sock:/var/run/docker.sock \ ubuntu:22.04 sleep
infinity
```

명령어 완벽 해설 (1/3)

`docker run`

신규 도커 상자(컨테이너) 생성을 지시합니다.

`-d` (Detach 모드)

터미널의 포그라운드를 차지하지 않고, 컨테이너를 **백그라운드에서 조용히 상시 실행**하도록 설정합니다.

`--name local-ubuntu-runner`

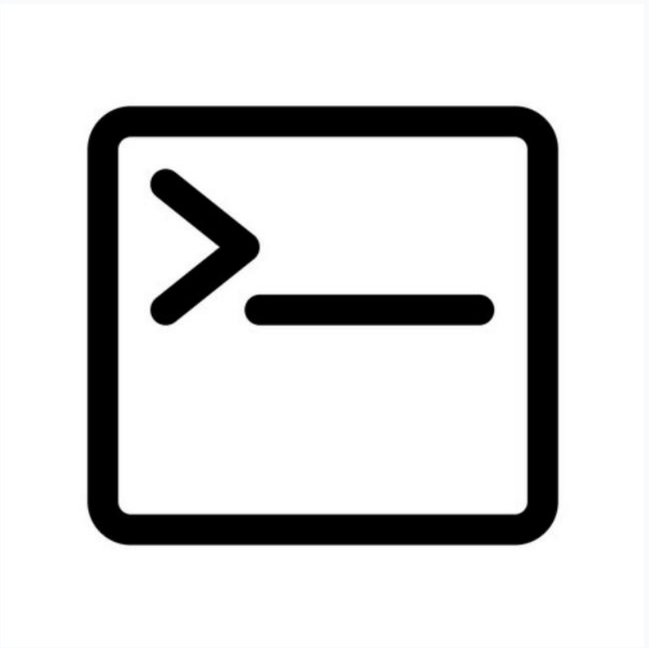
운영 및 식별 편의성을 위해 해당 상자의 명칭을 'local-ubuntu-runner'로 강제 지정합니다.

명령어 완벽 해설 (2/3)

`-v docker.sock`

본 실습의 핵심인 **DooD (Docker-out-of-Docker)** 기술입니다.

호스트(내 PC)의 도커 엔진 소켓을 컨테이너 내부에 공유 마운트합니다. 이를 통해 격리된 러너 안에서도 내 PC의 도커에게 새로운 서버 구동 명령을 하달할 수 있습니다.



명령어 완벽 해설 (3/3) - 왜 sleep infinity인가?

💡 핵심 원리: 컨테이너의 생명주기

일반적으로 `docker run -it ubuntu bash`를 사용하지만, 이는 터미널 세션 종료 시 서버가 즉시 동반 종료되는 치명적 한계가 존재합니다.

도커는 **메인 프로세스(PID 1)가 종료되면 컨테이너의 런타임도 완전 종료되는 특성**이 있습니다. 순정 우분투에는 상주 데몬이 없습니다.

따라서 `sleep infinity` **(무한 대기)** 명령을 메인 프로세스로 강제 주입하여, 24시간 명령을 수신해야 하는 서버의 런타임을 영구적으로 보장합니다.

Step 2

컨테이너 접속 및 필수 환경 세팅

우분투 내부 쉘(Shell) 침투

백그라운드에서 상시 가동 중인 우분투 내부 인스턴스에 접근합니다.

```
    # exec: 컨테이너 내부로 명령 하달, -it: 터미널 환경 연결 docker exec -it local-ubuntu-runner bash
```

✅ **프롬프트가 root@컨테이너ID:/# 로 변경되었다면 진입 성공입니다!**

필수 패키지 설치 및 권한 부여

초기화 상태의 텅 빈 우분투에 GitHub Actions 구동용 기반 도구를 세팅합니다.

```
    # 패키지 목록을 최신화합니다. apt update # 필수 구동 도구(curl, tar, git, docker 엔진 등)를 설치합니다. apt install -y curl tar git docker.io sudo libicu-dev # [중요] 일반 계정이 호스트의 도커 소켓에 접근할 수 있게 조작 권한을 활짝 엽니다. chmod 666 /var/run/docker.sock
```

일반 유저(runner) 계정 생성 및 세션 전환

⚠ 주의: root 계정 사용 제한 정책

GitHub의 보안 정책에 의거, 최상위 관리자(root) 계정 기반의 Runner 실행은 원천 차단됩니다. 일반 사용자 계정 생성이 필수입니다.

```
# 'runner' 명칭의 신규 계정 생성 및 홈 디렉토리(-m) 동시 할당 처리 useradd -m runner # 생성된 신규 'runner' 계정으로 터미널 세션 전환 su - runner
```

프롬프트가 `runner@컨테이너ID:~$` 로 갱신되면 환경 세팅 완료입니다.

Step 3

GitHub Runner 다운로드 및 호스트 등록

빈 저장소 생성 및 설정 메뉴 이동

- GitHub에서 **빈 레포지토리** 신규 생성 (예: `canary_test`)
- 해당 레포지토리 최상단 **[Settings]** 탭 클릭
- 좌측 메뉴 **[Actions]** → **[Runners]** 진입
- 우측 상단 **[New self-hosted runner]** 클릭
- 설치 대상 OS **Linux**, Architecture **x64** 지정

Runner 아카이브 다운로드 및 압축 해제

GitHub 화면에 안내된 설치 명령어를 컨테이너 터미널(runner 계정)에서 수행합니다.

```
    # 1. 작업 전용 폴더를 생성하고 진입합니다. mkdir actions-runner && cd actions-runner # 2. GitHub 서버로부터 구동 바이너리 압축 파일을 다운로드합니다. curl -o actions-runner-linux-x64-2.313.0.tar.gz -L https://github.com/actions/runner/ ... # 3. 다운로드 완료된 아카이브의 압축을 해제합니다. tar xzf ./actions-runner-linux-x64-2.313.0.tar.gz
```

Runner 에이전트 연동 설정 및 가동 개시

저장소 주소 및 인증 토큰을 매핑하여 에이전트 데몬을 구동합니다.

```
    # 4. 내 레포지토리 연동 (대화형 질의 시 모두 Enter 키를 입력해 기본값 지정)
./config.sh --url https://github.com/계정/레포명 --token [나의_비밀_토큰] # 5. 에이전트 데몬 호스트 구동 개시! ./run.sh
```

🎧 Listening for Jobs

해당 메시지 출력 시 성공적으로 GitHub 대기 상태 진입 완료입니다. **(터미널 창 유지)**

Step 4

웹 서비스 코드 및 Nginx 프록시 구성

프로젝트 폴더 아키텍처 (Project Root)

내 PC(호스트)에 새 터미널을 열고 canary_test 디렉토리를 생성합니다. 이곳이 최상단입니다.

```
📁 canary_test (프로젝트 최상단) ├── 📁 nginx/ │ ├── 📄 default.conf │ └── 📄 Dockerfile ├── 📄 server.js ├── 📄 Dockerfile └── 📄 deploy.sh
```

Nginx 라우팅 설정: default.conf

nginx/default.conf 파일 작성 (트래픽 분산 프록시 역할 수행)

```
    # 파이프라인이 외부에서 동적으로 가중치를 변환시킬 설정 파일을 사전 참조합니다. include /etc/nginx/conf.d/upstream.inc; server { listen 80; # 80번 포트로 인입되는 클라이언트 접속을 수신합니다. location / { proxy_pass http://backend; # 'backend' 업스트림 그룹으로 트래픽 프록시 포워딩 proxy_set_header Host $host; # 클라이언트의 원본 호스트 IP를 온전히 전달합니다. } }
```

Nginx 빌드 구성: Dockerfile 및 에러 차단

nginx/Dockerfile 파일 작성

```
FROM nginx:alpine COPY default.conf /etc/nginx/conf.d/default.conf # [핵심 방어] 초기 구동 시 Green 서버 미존재에 따른 Nginx 기동 실패를 차단하기 위해 단일 등록 강제화 RUN echo "upstream backend { server app-blue:8080; }" > /etc/nginx/conf.d/upstream.inc
```

💡 `weight=0` 및 DNS 탐색 에러 차단 전략

초기엔 Green 인스턴스가 부재하므로, 존재하지 않는 서버 명칭(DNS)을 기입하거나 Nginx가 해석하지 못하는 `weight=0` 문법을 할당하면 프록시가 즉시 추락합니다.

Node.js 웹 애플리케이션: server.js

프로젝트 최상단(Root)에 응답을 반환할 런타임 서버 코드를 작성합니다.

```
const http = require('http'); const PORT = process.env.PORT || 8080; const COLOR = process.env.COLOR || 'blue'; const VERSION = 'V1'; // 💡 데모 관측 시 이 버전값을 V2로 상향 수정하여 파이프라인 트리거 발생 예정 http.createServer((req, res) => { res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' }); res.end(`[${VERSION}] 현재 응답 서버 식별: ${COLOR} (포트: ${PORT})\n`); }).listen(PORT, () => { console.log(`서버 가동 완료 (PORT: ${PORT})`); });
```

웹 애플리케이션 빌드 구성: Dockerfile

프로젝트 최상단에 Node.js 코드를 이미지로 포장할 Dockerfile을 선언합니다.

```
    # 1. 경량화된 Node 18 Alpine 운영체제를 뼈대 베이스 이미지로 채택합니다. FROM node:18-alpine # 2. 컨테이너 내부 런타임 작업의 기준 디렉토리를 정의합니다. WORKDIR /usr/src/app # 3. 호스트에 존재하는 server.js 소스코드를 컨테이너 내부 런타임 공간으로 복사합니다. COPY server.js . # 4. 해당 컨테이너가 가동될 때 기본적으로 수행될 서버 실행 명령어를 명시합니다. CMD ["node", "server.js"]
```

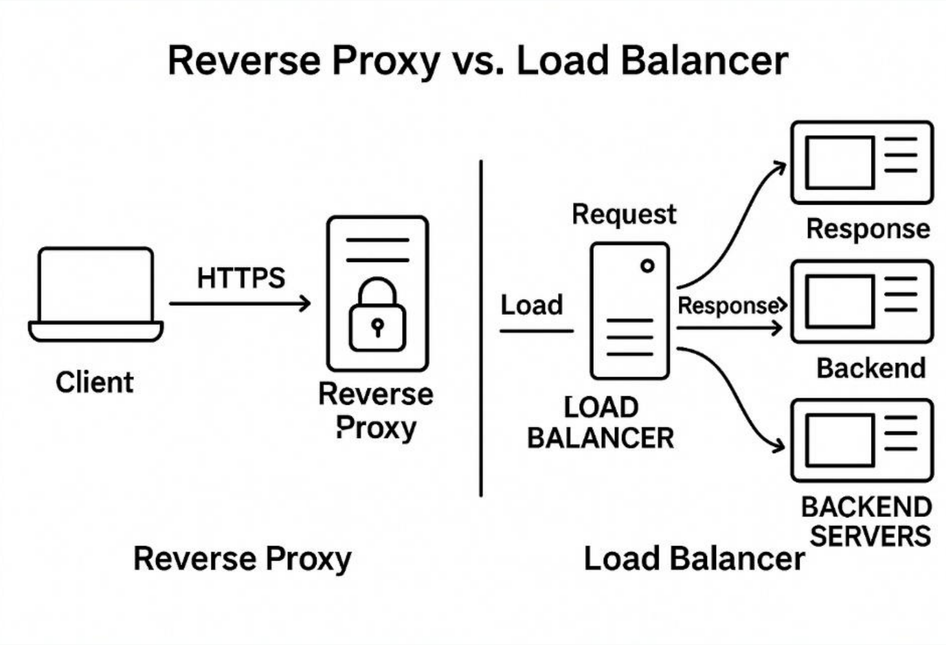
Step 5

카나리 스크립트 작성 및 알고리즘 딥다이브

deploy.sh 파이프라인 역할 개요

전체 배포 라이프사이클을 통제하는 **'오케스트레이션 두뇌(Brain)'** 스크립트입니다.

- 현재 활성 서버 인스턴스 교차 탐지 (Ping-Pong)
- 신규 컨테이너 빌드 및 가상 네트워크 결속
- Nginx 트래픽 라우팅 비율 무중단 동적 조절
- 자동 Health Check 기반 이상 유무 평가 및 롤백 조치
- 트래픽 배분 종료된 레거시 컨테이너 자동 파기



1. Nginx 라우팅 조종 리모컨 함수 구현

```
function update_nginx_weight() { local BLUE=$1; local GREEN=$2 # 할당 비율이 0 초과인 서버만 설정 문자열 명단에 편입합니다! (에러 방지) CONF="upstream backend { " [ "$BLUE" -gt 0 ] && CONF="${CONF} server app-blue:8080 weight=${BLUE}; " [ "$GREEN" -gt 0 ] && CONF="${CONF} server app-green:8081 weight=${GREEN}; " CONF="${CONF} }" # 호스트 밖에서 Nginx 컨테이너 내부로 설정 파일을 덮어쓰고 프록시를 리로드(Reload) 처리합니다. docker exec nginx-proxy sh -c "echo '$CONF' > /etc/nginx/conf.d/upstream.inc" docker exec nginx-proxy nginx -s reload }
```

💡 **`-gt 0` 의 스마트 방어 기제:** Nginx는 `weight=0` 인가 시 정지됩니다. 할당 비율이 0일 경우 명단에서 완전히 배제시켜 프로세스 정지를 막아냅니다.

2. 활성 서버 교차 탐지 (Ping-Pong 패턴)

```
    # 1. 호스트 내 app-blue 명칭 컨테이너의 가동 생존 여부 검색 IS_BLUE=$(docker ps -q -f name="^app-blue$") # 검색 결과 존재(Blue 가동 중) 시, 신규 배포를 수행할 타겟 서버는 Green으로 교차 스위칭 할당 if [ -n "$IS_BLUE" ]; then CURRENT="blue"; TARGET="green"; TARGET_PORT=8081; TARGET_COLOR="🟢 GREEN" else CURRENT="green"; TARGET="blue"; TARGET_PORT=8080; TARGET_COLOR="🔵 BLUE" fi
```

배포 파이프라인 수행 시마다 기존 활성 인스턴스와 신규 타겟 인스턴스의 위치가 탁구공처럼 교차(Ping-Pong) 전환 지정됩니다.

3. 신규 빌드 및 타겟 컨테이너 네트워크 결속

```
    # 2. 최신 릴리즈 소스코드를 기반으로 신규 도커 이미지를 구워냅니다. docker build -t my-canary-app . # 3. 이전 배포 실패 등 잔재로 남아있는 타겟 컨테이너 찌꺼기 존재 시 사전 강제 파기 조치 docker rm -f app-$TARGET 2>/dev/null # 4. 신규 컨테이너를 가상 네트워크망(canary-net)에 결속시켜 환경 변수 기반 구동 실행 docker run -d --name app-$TARGET --network canary-net -e PORT=$TARGET_PORT -e COLOR="$TARGET_COLOR" my-canary-app
```

4. 컨테이너 헬스체크(Health Check) 및 롤백

```
    # 5. 서버 내부 런타임 부팅을 위한 최소 대기 시간 부여 후, 프록시망 내부 접근 시도
sleep 5 RESPONSE=$(docker exec nginx-proxy sh -c "wget -qO-
http://app-$TARGET:${TARGET_PORT}") # 수신된 응답 문자열 패킷 미존재 시 서비스 불가로 판단, 즉각 컨테이너 파기(Rollback) 수행 if [ -z "$RESPONSE" ]; then echo "❌ 헬스체크 실패! 신규 서버 통신 접근 불가. 즉시 롤백 처리합니다." docker rm -f app-$TARGET exit 1 fi
```

실제 사용자 트래픽을 포워딩하는 주체인 **Nginx 내부 가상망 시야(DNS Lookup)**에서 신규 서버의 정상 응답 유무를 찌르며 검증하는 가장 정확도 높은 방식입니다.

5. 카나리 트래픽 점진적 전환 페이즈 (Stages)

```
    # 1단계: 신규 버전에 카나리 10% 트래픽 최초 할당 후 시스템 지표 15초 관측 대기 if [ "$TARGET" == "green" ]; then update_nginx_weight 90 10; else update_nginx_weight 10 90; fi; sleep 15 # 2단계: 구/신 서버 50:50 균등 분할로 트래픽 상향 전환 인가 후 15초 추가 모니터링 관측 if [ "$TARGET" == "green" ]; then update_nginx_weight 50 50; else update_nginx_weight 50 50; fi; sleep 15 # 3단계: 신규 릴리즈 버전 100% 완전 라우팅 전환 확정 달성 if [ "$TARGET" == "green" ]; then update_nginx_weight 0 100; else update_nginx_weight 100 0; fi # 라우팅 트래픽 배분 목적이 완전 종료된 이전 버전(Legacy) 컨테이너 인스턴스 파기 docker rm -f app-$CURRENT
```

타임라인 뷰: 점진적 트래픽 전환 시각화

1

Start

Health Check OK

2

Phase 1

10% Target

3

Phase 2

50% Target

4

Phase 3

100% Target

5

Finish

rm Legacy Server

Step 6

GitHub Actions 배포 지시서 (deploy.yml) 작성

배포 지시서 작성 (1/3) – 이벤트 작동 트리거

프로젝트 최상단 폴더에 .github/workflows/deploy.yml 파일을 생성합니다.

```
name: CI/CD Canary Deployment on: push: branches: [ "main" ]
```

트리거 동작 조건: main 브랜치에 코드가 Push(병합)되는 이벤트가 감지될 때마다 파이프라인 프로세스가 자동으로 기동되도록 선언합니다.

배포 지시서 작성 (2/3) - 러너 할당 인프라 지정

```
    jobs: canary-deploy: # GitHub 클라우드 인프라가 아닌, 앞서 구축한 '내 PC의 로컬 러너' 환경 직접 실행 지정 runs-on: self-hosted
```

실행 인프라 할당: 앞 단계에서 백그라운드 구동 환경에 연동시켜둔 `local-ubuntu-runner` 인스턴스(에이전트)가 이 파이프라인의 작업을 독점적으로 수신받아 처리하게 됩니다.

배포 지시서 작성 (3/3) - 상세 실행 스텝 정의

```
    steps: - name: 📥 소스코드 형상 최신화 Checkout uses:
actions/checkout@v3 - name: 🐦 카나리 무중단 배포 쉘 스크립트 실행 명령 하달
run: bash ./deploy.sh
```

세부 실행 단계: GitHub 원격 저장소에 반영된 최신 소스코드를 러너 로컬 영역으로 복제한 후, 시스템 단에서 스크립트(deploy.sh)를 호출하여 무중단 배포를 시작합니다.

Step 7

초기 네트워크 기반 세팅 및 원격 Git 형상 연동

1. Docker 사용자 정의 가상 네트워크망 생성

```
    # 'canary-net' 이라는 명칭의 컨테이너 논리 격리망을 독립 개설합니다.
docker network create canary-net
```

컨테이너 전용 내부 DNS의 중요성 파악

컨테이너 파기 및 재생성 시 **할당 내부 IP 주소는 매번 난수로 동적 변경**됩니다. 사용자 정의 네트워크망 구성 시 도커 데몬이 전용 내부 DNS 해석기를 가동하여, 오직 **이름(app-blue) 호출만으로도 바뀐 IP를 지속적으로 맵핑 라우팅**합니다.

2. 초기 뼈대 서버 파이프라인 수동 기동

파이프라인이 안정적으로 트래픽을 분산시킬 수 있도록, 초기 뼈대 베이스라인 환경을 구축합니다.

```
    # 첫 번째 베이스 기준 앱(Blue) 이미지 빌드 및 가상망 조인 기동 수행! docker build -t my-canary-app . docker run -d --name app-blue --network canary-net -e PORT=8080 -e COLOR="🔵 BLUE" my-canary-app # 트래픽 분배를 전담할 리버스 프록시(Nginx) 이미지 빌드 및 가상망 조인 80포트 개방 기동! docker build -t my-nginx ./nginx docker run -d --name nginx-proxy --network canary-net -p 80:80 my-nginx
```

3. Git 형상 저장소 초기화 및 소스코드 Push

현재 내 PC 호스트에 구성되어 있는 로컬 폴더를 깃허브 빈 레포지토리와 형상 연동 처리합니다.

```
    # 현재 로컬 폴더를 Git 형상관리 대상 저장소 영역으로 완전 초기화 지정 git init git
add . git commit -m "파이프라인 초기 세팅 반영 및 베이스 앱 배포" # 원격지 URL 할당 연동
후 소스코드 푸시 전송 git branch -M main git remote add origin https://github.com/아
이디/레포명.git git push -u origin main
```

Troubleshooting

빈번한 오류 대응 (PAT Workflow Scope Error)

에러 증상: 원격지 Push 인증 거부 현상

터미널 `git push` 명령 인가 시, 권한 인증 오류로 소스코드 반영이 강력히 거절되는 현상입니다.

```
! [remote rejected] main → main
(refusing to allow a Personal Access Token to
create or update workflow
.github/workflows/deploy.yml without workflow
scope)
```

에러 원인: GitHub 보안 샌드박스 정책

해커 침입 방어 아키텍처

악의적인 해커가 불법 탈취한 일반 토큰만으로 타겟 시스템에 치명적인 배포 스크립트(`.github/workflows/` 폴더 파일)를 무단 주입 및 실행하는 대형 보안 사고를 차단하기 위한 안전 장치입니다.

따라서 해당 특수 폴더 내 이력을 Push 시도하려면, **기본적인 Repo 접근 권한 외에 추가적으로 `workflow` 라는 명칭의 특수 보안 권한 스코프가 명시적으로 부여된 토큰이 반드시 요구**됩니다.

오류 해결 프로세스 (1/2)

- GitHub 웹사이트 우측 최상단 내 프로필 클릭 직후 **[Settings]** 메뉴 접속 진행
- 좌측 사이드바 영역 최하단 **[Developer settings]** → **[Personal access tokens (classic)]** 항목으로 이동
- 우측 상단 위치한 **[Generate new token (classic)]** 인증 버튼 클릭
- Note 시스템 식별자 기입란에 임의 명칭(예: Deploy Token) 할당 작성
- **(핵심적인 조치!)** 권한 허용 부여 체크박스 목록 중 `repo` 항목과 `workflow` 항목 2가지 동시 지정
- 최하단 Generate token 등록 버튼 클릭 후 화면에 표출되는 발급 문자열 토큰 복사

오류 해결 프로세스 (2/2)

복사해 둔 특수 인증 권한 포함 토큰을, 작업하던 터미널 로컬 Git Remote 연결 주소 내에 강제 덮어쓰기 적용합니다.

```
# 기존 맵핑 연결 주소를, 신규 복사한 토큰 데이터가 포함 결합된 완벽한 신규 주소로 치환 덮어쓰기 git remote set-url origin https://<신규발급토큰문자열>@github.com/아이디/레포명.git # 특수 스코프가 정상 인가 처리된 권한을 토대로 로컬 소스코드 푸시 재시도 수행! git push -u origin main
```

✅ **권한 인증 에러 문구 소거 완료 및 정상적인 형상 이관 시스템 승인 확인!**

The Demo

실시간 무중단 카나리 배포 트래픽 전환 시각적 관측

배포 검증 테스트 환경 세팅 전제 조건

- **초기 기준 상태 관측:** 웹 브라우저 주소창에 http://localhost 입력 후 접속 화면 유지 대기
- 화면 출력 텍스트가 명확하게 [V1] 🔵 BLUE 상태를 가리키는지 1차 육안 검수
- **릴리즈 버전 식별자 조작:** 호스트 에디터로 server.js 소스코드 파일 내용 직접 편집

```
    // 파일 내 기입된 V1 상숫값 문자열을 V2 명칭으로 상향 조작 변경 후 최종 저장 조치 이행합니다. const VERSION = 'V2';
```

V2 릴리즈 버전 신규 파이프라인 트리거 발생

수정 반영 조치된 코드를 현재 로컬의 커밋 이력으로 최종 확정짓고, GitHub 원격 저장소지로 소스를 밀어올려 자동화 파이프라인 프로세스를 즉각 가동합니다.

```
git add . git commit -m "V2 신규 릴리즈 버전 카나리 배포 자동화 파이프라인 검증 개시" git push origin main
```

무중단 실시간 트래픽 분산 비율 직접 관측

GitHub **[Actions]** 탭 구동 현황 로그 생성 확인 직후,

`http://localhost` **브라우저 화면에서 F5(새로고침) 버튼을 연속 입력하여, 응답 밸런싱의 동적인 변화 추이를 시각적으로 관측합니다.**

👀 페이즈(Phase)별 타임라인 관전 포인트

[관측 초기 15초]: 전체 트래픽 중 대략 10번에 1번 희소 빈도로 [V2] 🟢 GREEN 화면 단발성 무작위 표출

[관측 중기 15초]: Blue와 Green 응답이 50:50 확률 지표 기반으로 완전히 반반씩 교차 수신

[종료 직후]: 로직이 완료된 시점 이후에는 아무리 새로고침을 거듭해도 100% [V2] 🟢 GREEN 단일 응답만 표출

무중단 파이프라인 배포 완벽 구축! 🎉

복잡한 인프라 구성 지식 없이도, 순수 Docker 명령어와 GitHub 연동을 활용한 로컬 무중단 카나리 배포 아키텍처 구축을 성공적으로 마스터하셨습니다.

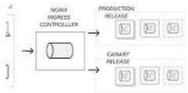
Q & A 질의 응답

긴 가이드 세션 경청해 주셔서 대단히 감사합니다. 본 인프라 관련 추가 논의를 진행하겠습니다.

Image Sources

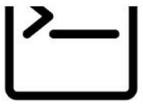
https://static.vecteezy.com/system/resources/thumbnails/067/345/142/small/canary-bird-line-art-logo-vector.jpg

Source: www.vecteezy.com



https://earthly.dev/.netlify/images?url=/blog/assets/images/canary-deployment-in-k8s/laz6hB1.png

Source: earthly.dev

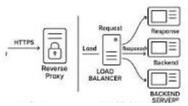


https://cdn.vectorstock.com/i/500p/57/45/terminal-icon-line-isolated-on-white-background-vector-45285745.jpg

Source: www.vectorstock.com

https://cdn.vectorstock.com/i/1000v/99/40/lock-icon-on-white-background-vector-20779940.jpg

Source: www.vectorstock.com



https://miro.medium.com/v2/resize:fit:1400/0*PtqpZLsiGptfvdao.png

Source: medium.com